Creating a Lightweight, Capable Convolutional Network for CIFAR Image Classification

Ioannis K. Antzoulis

East Stroudsburg University of Pennsylvania

I. INTRODUCTION

Convolutional neural networks (CNNs) have become one of the most widely utilized approaches to image classification tasks. The effectiveness of convolutions and pooling are well-documented, but due to the variability in the configuration of CNNs, there is no one correct way to construct a model. For example, VGG provides an unprecedented level of depth oftentimes requiring days to train on modern hardware. Residual neural networks (ResNets), however, take advantage of shortcuts to occur between layers, speeding up training and addressing the vanishing gradient problem. Therefore, modern techniques can be utilized to create a simple, lightweight network capable of classifying CIFAR-10 and CIFAR-100 images effectively.

Despite the constant advancements in image classification tasks and network architecture, some modern networks cannot be trained in an amount of time that justifies their complexity. For example, VGG was recognized for its unparalleled depth in ImageNet classification, but the performance gains could require up to 2-3 weeks to train on a system containing four GPUs [1, pp. 5]. These benefits are not worth the time, energy, or computational costs that are required to obtain them, however. Thus, it was important to ensure my network could be trained in 1-3 hours with a single discrete GPU.

Designing a neural network model requires that thought and consideration be given to all aspects of the network. Whether it is the size of convolutional layers or the choice in activation function, every design decision has an impact on the model's performance. I discovered that it is important to spend an equal amount of time on the foundational decisions of the model and fine-tuning hyperparameters to improve the network's performance.

This paper will detail the creation of a lightweight, capable network designed for classifying images of the CIFAR-10 and CIFAR-100 datasets. In Section II, background

information on convolutional networks, notable network architectures, and techniques will be provided. Section III includes the process of creating the model, code excerpts, and explanations of important design decisions. Tests will be conducted using various optimization algorithms, and the results will be reported in Section IV in addition to potential improvements to the model. Finally, Section V provides a brief summary of the paper in the form of a conclusion.

II. BACKGROUND

Convolutional networks are a class of deep learning networks that use convolutions to extract important features from the input data. Convolution works by computing the dot product between portions of the original input image and a matrix known as a *filter*. The process repeats until the entire input image has been convolved: the result of this process is called a *feature map*. During the training process, the network will begin to "learn" the values of these filters and begin to make better predictions as the number of filters increases. Different feature maps can be created using the same source image and are dependent on the filter used to create them. Convolutions are the key to many of the improvements seen in modern machine learning tasks, but there are many new techniques that can further improve network performance. One such technique is dropout.

In 2013, Google patented dropout [2], a technique that deactivates random nodes in a layer at a specified rate. For example, if a dropout rate of 0.5 is applied to a fully connected layer with 256 units, half of the layer's nodes will be shut down and the remaining half will be used. This method has been shown to reduce overfitting and has become standard in many modern CNN architectures.

Batch normalization, proposed by Ioffe and Szegedy in 2015 [3], is another technique that has become prevalent in recent years. When applied in a network, batch normalization

normalizes a previous layer's activation inputs before they are passed to a following layer. Its benefits are observable during testing; however, the reasons for its effectiveness are still contested. It was originally stated that batch normalization reduces the amount of internal covariate shift, or the changes made to activations as a result of the changes in a network's parameters [3, pp. 2]. For deep networks, these changes can be significant. Others, however, believe that batch normalization stabilizes training by smoothing the optimization and loss landscapes [4].

Along with uncertainty of how batch normalization provides its benefits, there has been a great deal of conflicting research regarding its ideal placement in a network. Ioffe and Szegedy explain the reason for applying it before the layer's activation: "We add the BN transform immediately before the nonlinearity, by normalizing $x = Wu+b$ … since u is likely the output of another nonlinearity, the shape of its distribution is likely to change during training, and constraining its first and second moments would not eliminate the covariate shift. [3, pp. 5]" Despite this claim, the author of Keras revealed that even the creator of batch normalization has reversed its order preferring to include it after the nonlinearity [5]. Dropout and batch normalization are two highly effective techniques for improving image classification performance in CNNs. These techniques are not useful if they are not properly utilized in networks, so it is imperative to discuss two landmark CNN architectures and how they contributed to modern CNN design.

Originally proposed in 1998 by LeCun et al. [6], LeNet-5 has become a classic network architecture. It contains two sets of convolutional layers and pooling layers followed by a final convolutional layer. Then, the inputs are flattened and fed into two fully connected layers and an output activation. At the time, it achieved exceptional performance in classifying handwritten

digits of the MNIST dataset and character recognition. Many of the principles established in LeNet-5 would influence later CNN architectures including AlexNet.

In 2012, AlexNet was proposed by Krizhevsky et al [7]. Several decades had passed since the creation of LeNet-5 including an increase in computing power and network task complexity. This led to major advancements in machine learning tasks. The introduction of the ImageNet dataset gave birth to many of the deep, robust networks of today. Under the team name SuperVision, AlexNet won the 2012 ImageNet Large Scale Vision Recognition Challenge by a large margin [8].

In 2009, three years before the creation of AlexNet, Krizhevsky created two datasets: CIFAR-10 and CIFAR-100 [9]. CIFAR-10 contains 60,000 images containing animals and vehicles. This set is divided into five 10,000 image batches for training and a single 10,000 image batch for validation [10]. CIFAR-100 follows the same structure as CIFAR-10, but there are 100 individual classes that can be grouped into twenty superclasses.

III. METHODS

One of the first steps in creating a CNN is analyzing the complexity of the problem to be solved. Classifying ImageNet images requires a much deeper network than CIFAR classification, so it is essential that a network is tailored to the chosen dataset. Fortunately, the design practices created in LeNet-5 and expanded upon by AlexNet continue to remain relevant today.

When first creating the network, a series of tests were performed to discover the limits the network would have. For example, very early in the testing process it became apparent that convolutional layers containing less than 32 filters and more than 512 would not be optimal for the network. This preliminary set of tests provided an idea of what baseline performance I could expect. A similar set of tests were performed to determine the ideal number and size of fully

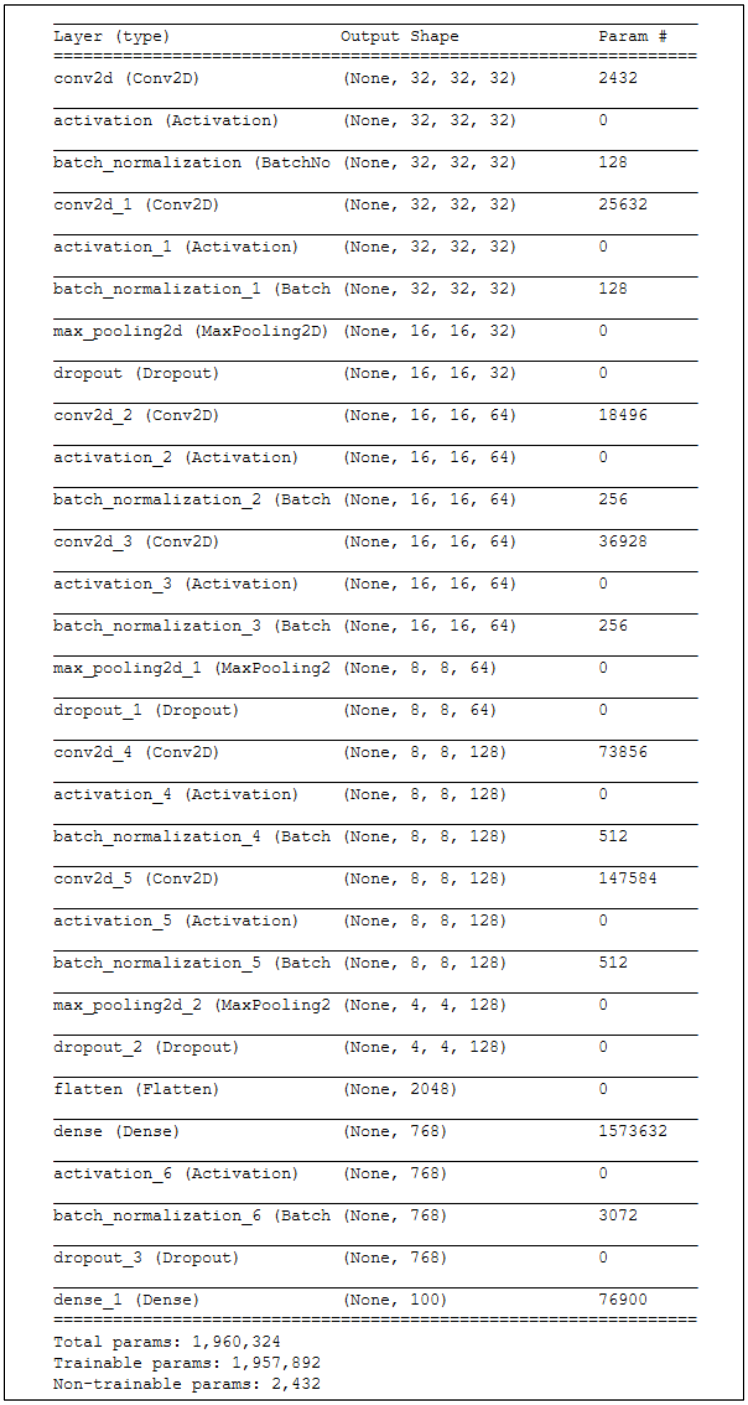
connected layers for the network. Fully connected layers of size 1024, 768, 512, and 256 units were all evaluated in one- and two-layer configurations.

Block	Layer	Name	Size
1	1	Convolution-ReLU	5x5, 32
	2	Convolution-ReLU	5x5, 32
	3	Max Pooling	2x2
2	4	Convolution-ReLU	3x3, 64
	5	Convolution-ReLU	3x3, 64
	6	Max Pooling	2x2
3	7	Convolution-ReLU	3x3, 128
	8	Convolution-ReLU	3x3, 128
	8	Max Pooling	2x2
	9	Fully Connected	768
	10	Output (Softmax)	100

Table 1. Detailed overview of the CIFAR-100 network structure.

The structure of my final network heavily draws from the principles pioneered by LeCun et al. in LeNet-5 and later expanded upon by Krizhevsky et al. in AlexNet. Table 1 contains the structure of my network. The first two convolutional layers of the network contain a kernel size of 5x5 to help extract larger features from the images; subsequent convolutional layers contain a 3x3 kernel. A 5x5 kernel is beneficial during earlier stages in training because larger features can be extracted before proceeding to deeper layers. After the final max pooling layer, the inputs are flattened and fed into a single fully connected layer. The CIFAR-10 configuration contains a single fully connected layer containing 512 units, and the CIFAR-100 configuration contains one 768-unit layer.

In addition to what is seen above, Fig. 1 and 2 show the model's information in greater detail. Figure 1 contains a summary of the model including the layers, their output shapes, and the number of parameters for each layer.



Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 32, 32, 32)	2432
activation (Activation)	(None, 32, 32, 32)	0
batch_normalization (BatchNo	(None, 32, 32, 32)	128
conv2d_1 (Conv2D)	(None, 32, 32, 32)	25632
activation_1 (Activation)	(None, 32, 32, 32)	0
batch_normalization_1 (Batch	(None, 32, 32, 32)	128
max_pooling2d (MaxPooling2D)	(None, 16, 16, 32)	0
dropout (Dropout)	(None, 16, 16, 32)	0
conv2d_2 (Conv2D)	(None, 16, 16, 64)	18496
activation_2 (Activation)	(None, 16, 16, 64)	0
batch_normalization_2 (Batch	(None, 16, 16, 64)	256
conv2d_3 (Conv2D)	(None, 16, 16, 64)	36928
activation_3 (Activation)	(None, 16, 16, 64)	0
batch_normalization_3 (Batch	(None, 16, 16, 64)	256
max_pooling2d_1 (MaxPooling2	(None, 8, 8, 64)	0
dropout_1 (Dropout)	(None, 8, 8, 64)	0
conv2d_4 (Conv2D)	(None, 8, 8, 128)	73856
activation_4 (Activation)	(None, 8, 8, 128)	0
batch_normalization_4 (Batch	(None, 8, 8, 128)	512
conv2d_5 (Conv2D)	(None, 8, 8, 128)	147584
activation_5 (Activation)	(None, 8, 8, 128)	0
batch_normalization_5 (Batch	(None, 8, 8, 128)	512
max_pooling2d_2 (MaxPooling2	(None, 4, 4, 128)	0
dropout_2 (Dropout)	(None, 4, 4, 128)	0
flatten (Flatten)	(None, 2048)	0
dense (Dense)	(None, 768)	1573632
activation_6 (Activation)	(None, 768)	0
batch_normalization_6 (Batch	(None, 768)	3072
dropout_3 (Dropout)	(None, 768)	0
dense_1 (Dense)	(None, 100)	76900

Total params: 1,960,324
Trainable params: 1,957,892
Non-trainable params: 2,432

Figure 1. CIFAR-100 model summary.

Figure 2 contains the implementation of the model in Keras. In addition to what is shown in Table 1, dropout with a rate of 0.25 is applied after each max pooling layer and with a rate of

0.5 for each fully connected layer. Batch normalization is also applied after the activations for each convolutional and fully connected layer.

```
weight_decay = 1e-4

model = Sequential()
model.add(Conv2D(32, (5, 5), kernel_regularizer=l2(weight_decay), padding = 'same',
          input_shape = train_images.shape[1:]))
model.add(Activation('relu'))
model.add(BatchNormalization())
model.add(Conv2D(32, (5, 5), kernel_regularizer=l2(weight_decay), padding = 'same'))
model.add(Activation('relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size = (2, 2), strides = (2, 2)))
model.add(Dropout(0.25))

model.add(Conv2D(64, (3, 3), kernel_regularizer=l2(weight_decay), padding = 'same'))
model.add(Activation('relu'))
model.add(BatchNormalization())
model.add(Conv2D(64, (3, 3), kernel_regularizer=l2(weight_decay), padding = 'same'))
model.add(Activation('relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size = (2, 2), strides = (2, 2)))
model.add(Dropout(0.25))

model.add(Conv2D(128, (3, 3), kernel_regularizer=l2(weight_decay), padding = 'same'))
model.add(Activation('relu'))
model.add(BatchNormalization())
model.add(Conv2D(128, (3, 3), kernel_regularizer=l2(weight_decay), padding = 'same'))
model.add(Activation('relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size = (2, 2), strides = (2, 2)))
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(768))
model.add(Activation('relu'))
model.add(BatchNormalization())
model.add(Dropout(0.5))

model.add(Dense(100, activation = 'softmax'))
```

Figure 2. Implementation of CIFAR-100 model in Keras

There are several optimizer algorithms available to use with their own benefits. SGD and Adam were chosen because they both offer different approaches to updating network parameters. SGD is a proven optimizer that benefits from fine-tuning, whereas Adam updates SGD to utilize an adaptive learning rate. Adam was tested with a learning rate of 0.01 and SGD with an initial learning rate of 0.1 and Nesterov momentum set to 0.9. Due to the behavior of Adam's adaptive

learning rate, decay is not necessary. SGD, however, greatly benefits from learning rate decay. After every 25 epochs, the learning rate is set to decrease by 0.5 (Figure 3).

```
def lr_schedule(epoch):
    lr_initial = 0.1
    drop = 0.5
    drop_interval = 25.0

    # Formula to compute the learning rate based on defined values
    lr = lr_initial * math.pow(drop, math.floor((1 + epoch)/drop_interval))
    return lr
lrate = LearningRateScheduler(lr_schedule)
```

Figure 3. Learning rate decay schedule for SGD optimizer.

As seen in Figure 4, preprocessing was applied to both the training and validation data, and data augmentation was performed on the training data in the form of rotation, horizontal flips, and shifts on the horizontal and vertical axes. To preprocess the images, they are first converted to 32-bit float. Then, the mean and standard deviation of the training set are computed, and finally, for each set of images, the mean is subtracted, and the result is divided by the standard deviation. Because the validation set is a subset of images found in the training data, the mean and standard deviation can be computed on the training images exclusively.

```
train_images = train_images.astype('float32')
test_images = test_images.astype('float32')

# Mean and standard deviation are computed on training images
mean = np.mean(train_images,axis=(0,1,2,3))
std = np.std(train_images,axis=(0,1,2,3))

# Training and testing images are normalized
train_images = (train_images-mean)/(std)
test_images = (test_images-mean)/(std)

datagen = ImageDataGenerator(
    featurewise_center = False,
    featurewise_std_normalization = False,
    rotation_range = 15,
    width_shift_range = 0.1,
    height_shift_range = 0.1,
    fill_mode = 'nearest',
    horizontal_flip = True)
datagen.fit(train_images)
```

Figure 4. Preprocessing method and data augmentation values for dataset.

IV. RESULTS

A. *CIFAR-100*

For the CIFAR-100 dataset, my model was trained for 125 epochs with the default training/validation split of 50,000 training images and 10,000 validation images. A batch size of 128 was used, although this value can be altered with little impact on performance. SGD and Adam, the two optimizers detailed in the previous section, were each tested several times to establish a level of expected performance from the model.

After 125 epochs, SGD reached a maximum training accuracy of 70.94% and a validation accuracy of 65.46%. As seen in Figure 5, overfitting occurs in the latter half of training as the model begins to learn training features that are not reinforced by the validation stage. The use of L2 kernel regularization, data augmentation, dropout, and batch normalization were effective in reducing the amount of overfitting in the network but did not eliminate it completely. At the point of greatest variance, the training and validation accuracies vary by roughly 6%. This difference implies that the length of training was longer than necessary and could be shortened.

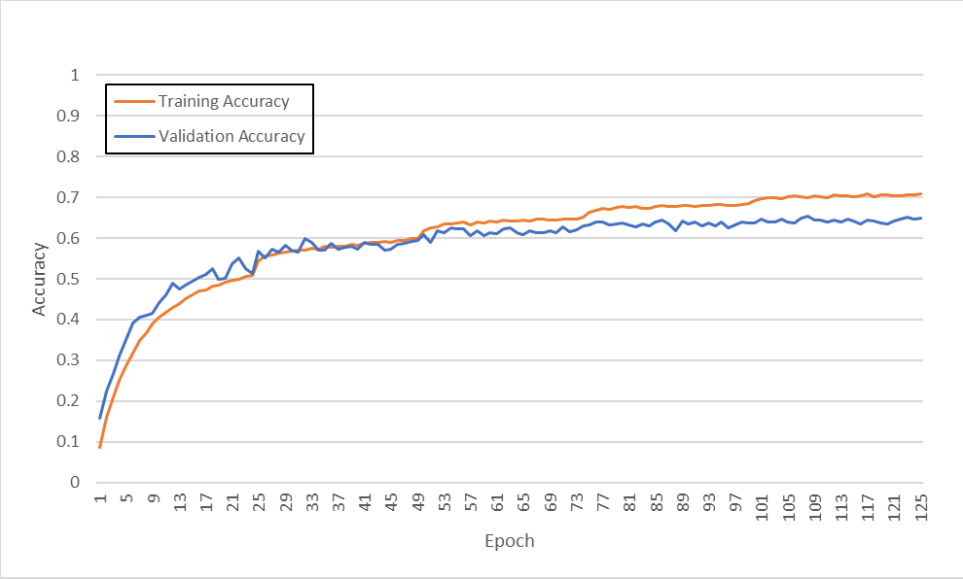


Figure 5. Training and validation accuracy graph for a CIFAR-100 SGD optimizer test.

Adam reached top training and validation accuracies of 69% and 64.74%, respectively. Because Adam utilizes an adaptive learning rate, a learning rate decay schedule was unnecessary for this optimizer. Despite the lack of learning rate decay, Adam performs within 1% of SGD. In addition to accuracy, it is important to interpret the loss values to further evaluate the model's performance.

As seen in Figure 6, the model shows consistent behavior throughout training. Plotting the loss values of two tests also helps further analyze the impact of different optimization algorithms. These results show that Adam produces a more significant reduction in both training and validation loss early in training while SGD's learning rate decay provides an advantage in the later stages of training. Additionally, the effects of learning rate decay can be seen at every 25 epochs on the SGD training loss value.

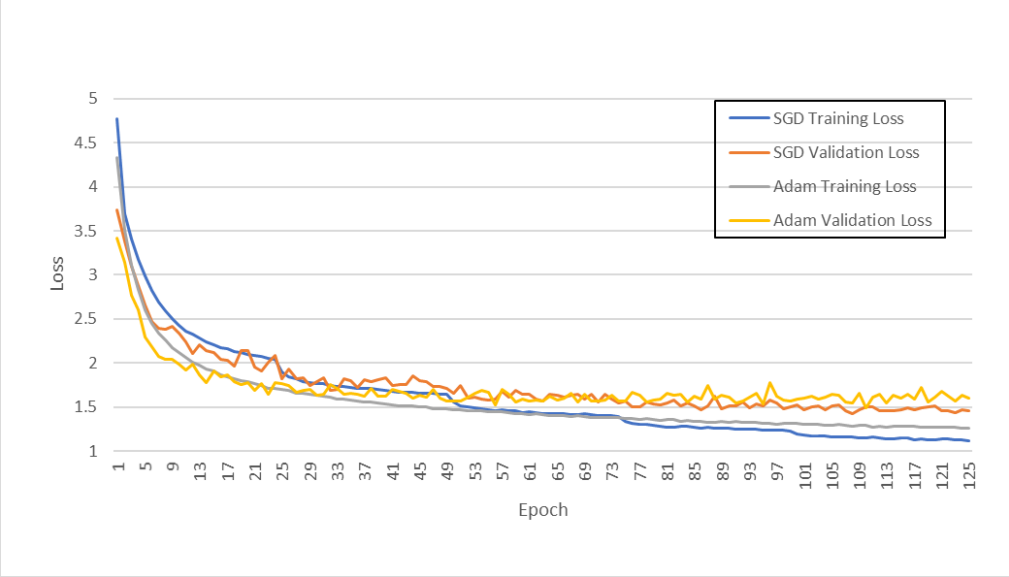


Figure 6. Loss graph for CIFAR-100 SGD and Adam optimizer tests.

B. CIFAR-10

Although I had originally designed my model specifically for the CIFAR-100 dataset, I began to evaluate it on CIFAR-10. What I found is that the network performed exceptionally

well with very minor alterations. Instead of using a 768-unit fully connected layer, the size was decreased to 512 units. Upon testing, it also became clear that CIFAR-10 did not require the same length of training as CIFAR-100, so the training length was decreased from 125 epochs to 100 epochs.

The results appear to show a very different relationship between the length of training and amount of overfitting. If we recall the results of the CIFAR-100 model, overfitting began to occur when the validation stage was not benefiting from the improvements made during the training stage. When the model was trained on CIFAR-10, the amount of overfitting decreased significantly. The CIFAR-10 results show that both the training and validation stages are learning at a similar rate (Figure 7), reaching a top training accuracy of 89% and validation accuracy of 88.88%. This relationship indicates that the network is performing to its full capabilities, and there is very little learning that is not carried between the training and validation stages.

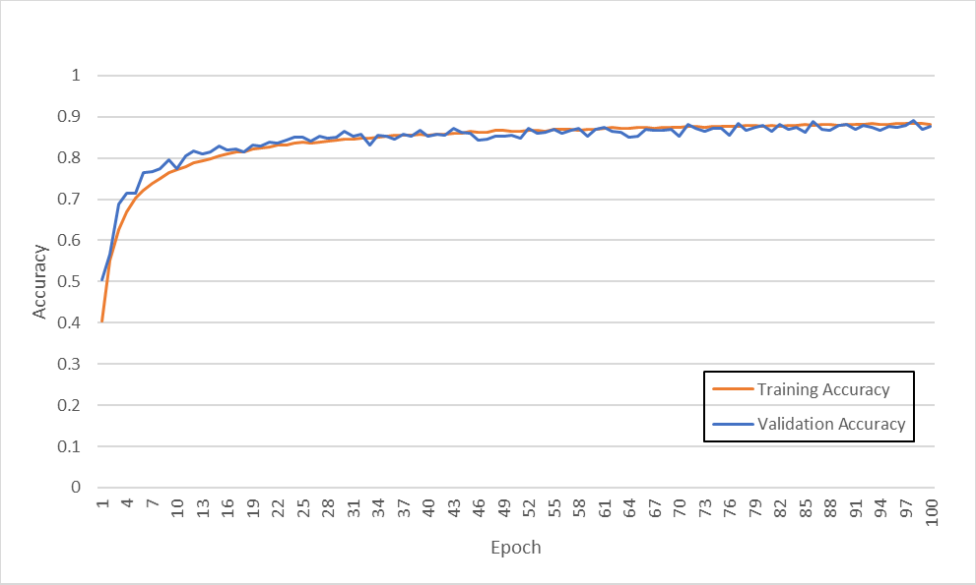


Figure 7. Training and validation accuracy graph for CIFAR-10 Adam optimizer test.

C. Future Work

Despite reaching most of the goals I had initially set for the model, there remains potential for improvements that can further improve performance. One feature that may reduce overall training time is early stopping. Early stopping can be implemented to track values, most often the validation loss, and end training when the metric does not improve. In addition to eliminating unnecessary training time, early stopping will significantly reduce the amount of overfitting because training can be ended at the first signs of overfitting.

In addition to loss and accuracy, k-fold cross-validation is one of the best metrics for model assessment. Cross-validation is not available within Tensorflow, but Scikit-Learn may be used to provide this functionality. Using Scikit-Learn's cross-validation would require reworking code to be compatible with the new library, but the benefits would allow for better assessment of the model and may provide valuable information when enhancing the model.

Finally, I feel that it would be beneficial to perform hyperparameter optimization on the model. Manual optimization is time consuming and less precise than running automated tests. Some of the methods available to optimize hyperparameters include random search, grid search, and Bayesian optimization. Scikit-Learn provides classes for random search and grid search, and Bayesian optimization may be implemented if needed.

V. CONCLUSION

Convolutional neural networks (CNNs) are a class of deep networks that excel in image classification tasks. With two major components, convolutional and pooling layers, a network can be designed to fit almost any task. Thus, it was my goal to design a lightweight, capable network for classifying images of the CIFAR-10 and CIFAR-100 datasets.

Originally, I intended to focus my model solely on the CIFAR-100 dataset. Beginning with a simple model, I began to learn what changes reflected positively in my results and which ones needed tweaking. As I began to understand the dataset, my final design became clear. What followed was the application of various regularization techniques to reduce the amount of overfitting and final adjustments to various parameters. The final network performed well with top training and validation accuracies of 70.94% and 65.46%, respectively.

When the model was transferred to CIFAR-10, the results represented a significant improvement over the CIFAR-100 model. This improvement could be partly attributed to CIFAR-10 being a simpler dataset than its 100-class counterpart, but the results show less overfitting and more consistent improvements during training. This model reached a training accuracy of 89% and validation accuracy of 88.88%.

Overall, I am very pleased with my final model and how well it performed during testing. Two of my main goals for this project were to create a model that could be useful in day-to-day image classification tasks and to expand my knowledge of machine learning. Through this project, I studied landmark CNN architectures, gained proficiency using several Python machine learning libraries, and applied important machine learning techniques and concepts. In the future, I hope to further expand upon my work by incorporating new knowledge, libraries, and techniques into my skillset.

References

[1] K. Simonyan and A. Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition,” International Conference on Learning Representations 2015 (ICLR 2015), 10-Apr-2015. [Online]. Available: https://arxiv.org/abs/1409.1556

[2] “US9406017B2 - System and method for addressing overfitting in a neural network,” *Google Patents*. [Online]. Available: https://patents.google.com/patent/US9406017B2/en.

[3] S. Ioffe and C. Szegedy, “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift,” *arXiv.org*, 02-Mar-2015. [Online]. Available: https://arxiv.org/abs/1502.03167.

[4] S. Santurkar, D. Tsipras, A. Ilyas, and A. Madry, “How Does Batch Normalization Help Optimization?,” *arXiv.org*, 15-Apr-2019. [Online]. Available: https://arxiv.org/abs/1805.11604.

[5] https://github.com/keras-team/keras/issues/1802#issuecomment-187966878

[6] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-Based Learning Applied to Document Recognition,” Proceedings of the IEEE, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.

[7] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet Classification with Deep Convolutional Neural Networks,” Advances in Neural Information Processing Systems, vol. 25, Jan. 2012.

[8] “Large Scale Visual Recognition Challenge 2012 (ILSVRC2012),” ImageNet Large Scale Visual Recognition Competition 2012 (ILSVRC2012), 13-Oct-2012. [Online]. Available: http://www.image-net.org/challenges/LSVRC/2012/results.html.

[9] A. Krizhevsky, “Learning Multiple Layers of Features from Tiny Images,” University of Toronto, 2009. [Online]. Available: https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf

[10] A. Krizhevsky, “CIFAR-10 and CIFAR-100 datasets,” Department of Computer Science, University of Toronto. [Online]. Available: https://www.cs.toronto.edu/~kriz/cifar.html.